# System Design

This is a document describing the design of the NetarchiveSuite software

This document only describes the underlying design of the NetarchiveSuite software, i.e. it does not describe how to install, run, or use NetarchiveSuite, for that see the Installation Manual and the User Manual.

The first section gives an overview, and the remainder of the document gives more details about the design.

The code is available through the downloaded site or from our Subversion repository .

### Contents



### Audience

The reader is expected to be familiar with Java programming and have an understanding of the core issues involved in large-scale web harvesting. Having used Heritrix before is a definite plus, and an elementary understanding of SQL databases is required for some parts.


**Search manual**

Download as pdf


# Overall Systems Design

**Contents**

For a architectural overview see the NetarchiveSuite Overview page.

For more detailed information on the different modules, see the javadoc documentation.

# Settings

Settings are read using methods in the class dk.netarkivet.common.utils.Settings.The declaration of the settings themselves, however, have been moved to new settings classes in each of the modules. Every module except for the Deploy module have its own Settings class and a settings.xml file containing the default values for the module settings.

In addition to that, every plug-in is intended to declare its own settings inside itself, and be associated with an xml-file containing the default values of these settings placed in the same directory as the plugin itself. For more information, please refer to the Configuration Basics description in the Configuration Manual.

Associated with most of the NetarchiveSuite plug-ins, there are also factory classes that hides the complexity behind selecting the correct plugin according to the chosen settings. The names of these classes all ends on Factory, e.g. JMSConnectionFactory, RemoteFileFactory.

Almost all configuration of NetarchiveSuite is done through the main module Settings classes as for example the dk.netarkivet.common.Settings class. It provides a simple interface to the settings.xml file as well as definitions of all current configuration default settings. The settings.xml file itself is an XML file with a structure reminiscent of the package structure.

Settings are referred to inside NetarchiveSuite by their path in the XML structure. For instance, the storeRetries setting in arcrepositoryClient under common is referred to with the string `settings.common.arcrepositoryClient.storeRetries`. However, to avoid typos, each known setting has its path defined as a String constant in a module Settings class(e.g. HarvesterSettings), which is used throughout the code (also by deploy in checks of whether the settings are known). The module Settings class file also includes description of the setting in their javadoc.

To add a new general setting, the following steps need to be taken:

1) The Settings class should get a definition for the path of the setting. This String, although a constant, must mot be declared final since settings are initialised by a static initialiser in the class

2) Javadoc for the definition must including the path name of the setting as well as description of the setting.

3) All default settings.xml files must be updated (including those in the unit tests).

4) examples/settings_example.xml must be updated.

Note that there are no XML Schema to be updated, because the use of default settings means that a settings file does not need to be there, and in the case of settings for plug-ins we generally do not know which setting will be used.

Note also that the Configuration Manual includes a description of how deploy can be used to validate whether the settings in a settings file are valid (not necessarily exhaustive!).

# Localization

The NetarchiveSuite web pages are internationalized, that is they are ready to be translated into other languages. The default distribution contains a default (English) version and Danish, Italian, French and German versions, but adding a new language does not take any coding. All translatable strings are collected in five resource bundles, one for each of the five main modules mentioned above. The default translation files are `src/dk/netarkivet/common/Translations.properties`, `src/dk/netarkivet/archive/Translations.properties`, `src/dk/netarkivet/harvester/Translations.properties`, `src/dk/netarkivet/viewerproxy/Translations.properties`, and `src/dk/netarkivet/monitor/Translations.properties`.

To translate to a new language, first copy each of these files to a file in the same directory, but with _XX after Translations, where XX is the Unicode language code for the language you're going to translate into, e.g. if you're translating into Limburgish, use `Translations_li.properties`. If you're translating into a language that has different versions for different countries, you may need to use _XX_YY, where XX is the language code and YY is the ISO country code, e.g. `Translations_fr_CA.properties` for Canadian French. Then edit each of the new files to have your translation instead of the English translation for each line. Most of the important syntax should be evident from the original, but for details consult the XXX. According to the Java documentation (specifically the Javadoc of the Properties class) resource bundles should use iso-8859-1 with escaped Unicode for all other characters. It is good practice to use escaped Unicode for "all" non-ASCII characters as this results in files which are more-easily shared between different text-editing environments.

The translation has not been done throughout the code, only in the web-related parts. Thus log messages and unexpected error messages are in English and cannot be translated through the resource bundles.

# JMS Channels

**Contents**



### Placement in NetarchiveSuite software

Every channel is named after the set of applications instances that are expected to receive messages sent to it. All channel names are constructed privately in the `dk.netarkivet.distribute.ChannelID` class. To get a channel, you must use one of the public methods in `dk.netarkivet.distribute.Channels`.

### Channel Behavior

There are used to types of channels:

- **Queue** where only one listener takes a message from the queue. For example a queue where a request for doing a harvest is only received by one harvester.
- **Topic** where all listeners takes a copy of the message. For example a batch job which has to be executed by

all bitarchive applications.
The type of channel is not affecting the channel name directly. It is indicated by the Channel Prefix, since only channel names starting with ALL are topics, while the rest are queues.

The channel type for different queues is given in a table in the next section.

### Naming Conventions

The structure of channel names are as follows:

```
<Environment Name>_<Channel Prefix>_<Replica Annotation>[_<Machine>][_<Application Instance>]
```

- **Environment Name** value must be the same for all channels on all JVMs belonging to a single NetarchiveSuite installation. It identifies the environment for the queue, e.g.: "DEV", "TEST", "RELEASETEST", "CLOVER", "PROD" or initials of developer running personal environment for e.g. sanity tests. It is read from the setting `settings.common.environmentName` of the NetarchiveSuite installation.
- **Channel Prefix** is constructed by a convention for channel behavior together with denotation of the concerned application(s). For example **THE_SCHED** uses the convention **THE** for **SCHED** denoting the scheduler of the harvesting system. The scheduler is now embedded in the HarvestJobManager.

The conventions for channel behavior are:

- **THE** - communicate with singleton in the system.
- **THIS** - communicate with one uniquely identified instance of a number of distributed listeners.
- **ANY** - has all instances listening to the same queue.
- **ALL** - is used for topics only, i.e. topics are sent to all listeners on the channel.
- **Use Replica Id** will result in a channel name with the replica identifier in question (which must match possible replicas in settings). In case of no use of replica id the **COMMON** will be used.
- **Use IP Node** will result in a channel name with the IP address of the machine whether the application in question is placed. It is read directly from the running system.
  . In case of no use of IP Node, nothing will be written.
- **Use Application Instance Id** is used, if only one process is expected to listen to the queue. It will result in a channel name with an identification of the individual application instance. It consists of the application abbreviation (uppercase letters of application name) and the application instance identifier from the settings. It is read from the common settings: `settings.common.applicationName` and `settings.common.applicationInstanceId`.
  . In case of no use of Application Instance Id, nothing will be written.
- **Channel Type** is only here to complete the picture. Please refer to the section on channel behavior.
  Channel names are constructed as described in the below table (columns described after the table).

| Channel Prefix | Use Replica Id<br><br>(for Replica Annotation) | Use IP Node (for Machine) | Use Application Instance Identification | Channel Type | Example |
|---|---|---|---|---|---|
| THE_SCHED | No | No | No | Queue | PROD_THE_SCHED_COMMON |

| | | | | | |
|---|---|---|---|---|---|
| ANY_HIGHPRIORITY_HACO | No | No | No | Queue | PROD_ANY_HIGHPRIORITY_HACO_COMMON |
| ANY_LOWPRIORITY_HACO | No | No | No | Queue | PROD_ANY_LOWPRIORITY_HACO_COMMON |
| THIS_REPOS_CLIENT | No | Yes | Yes | Queue | PROD_THIS_REPOS_CLIENT_COMMON_130_226_258_7_HCA_HIGH |
| THE_REPOS | No | No | No | Queue | PROD_THE_REPOS_COMMON |
| THE_BAMON | Yes | No | No | Queue | PROD_THE_BAMON_TWO |
| ALL_BA | Yes | No | No | Topic | PROD_ALL_BA_TWO |
| ANY_BA | Yes | No | No | Queue | PROD_ANY_BA_TWO |
| THIS_INDEX_CLIENT | No | Yes | Yes | Queue | PROD_THIS_INDEX_CLIENT_COMMON_130_226_258_7_ISA |
| INDEX_SERVER | No | No | No | Queue | PROD_INDEX_SERVER_COMMON |
| MONITOR | No | No | No | Queue | PROD_MONITOR_COMMON |
| ERROR | No | No | No | Queue | PROD_ERROR_COMMON |
| THE_CR | Yes | No | No | Queue | PROD_THE_CR_THREE |

The examples are using values

- Environment name `PROD`
- Possible replica identifiers `ONE` or `TWO`
- IP on machine `130.226.258.7`
- Application instances
- `HCA_HIGH` (for HarvestControllerApplication with instance id `HIGH` )
- `ISA` (for IndexServerApplication with instance id `""` )

### Design Notes

Note that creation of channel names for the ANY xxx_HACO-queues are designed in a way so extension to more priorities is easy.

# JSP

**Contents**



The webpages in NetarchiveSuite are written using JSP (Java Server Pages) with Apache Jakarta I18N Taglib for internationalization. To support a unified look across pages from different modules, we have divided the pages into SiteSections as described in the next section. Any processing of requests happens in Java code before the web page is displayed, such that update errors can be handled with a uniform error page. Internationalization is primarily done with the taglib tags <fmt:message>, <fmt:date> etc.

The main feature of JSP is that ordinary Java (not JavaScript) can be used at server-side to generate HTML. The special tags <%...%> indicate a piece of Java code to run, while the tags <%=...> indicates a Java expression to run whose value will be inserted (as is, see escape mechanisms below) in the HTML. While it is possible to output to HTML from Java code using **out.print()**, it is discouraged as it a) is confusing to read, and b) does not allow for using taglibs for internationalization.

We use a number of standard methods defined in **dk.common.webinterface.HTMLUtils**. Of particular note are the following methods:

**generateHeader()**:: This method takes a **PageContext** and generates the full header part of the HTML page, including the starting <body> tag. It should always be used to create the header, as it also creates the menu and language selection links. After this method has been called, redirection or forwarding is no longer possible, so any processing that can cause fatal errors must be done before calling **generateHeader()**. The title of the page is taken from the **SiteSection** information based on the URL used in the request.

**generateFooter()**:: This closes the HTML page and should be called as the last thing on any JSP page.

**setUTF8()**:: This method must be called at the very start of the JSP page to ensure that reading from the request is handled in UTF-8.

**encode()**:: This encodes any character that is not legal to have in a URL. It should be used whenever an unknown

string (or a string with known illegal characters) is made part of a URL. Note that it is not idempotent, calling it twice on a string is likely to create a mess.

**escapeHTML()**:: This escapes any character that has special meaning in HTML (such as < or &). It should be used any time a unknown string (or a string with known special characters) is being put into HTML. Note that it is **not** ide mpotent: If you escape something twice, you get a horrible-looking mess.

**encodeAndEscape()**:: This method combines **encode()** and **escapeHTML()** in one, which is useful when you're putting unknown strings directly into URLs in HTML.

### The SiteSection system

Each part of the web site (as identified by the top-level menu items on the left side) is defined by one subclass of the SiteSection class. These sections are loaded through the <siteSection> settings, each of which connect one SiteSection class with its WAR file and the path it will appear under in the URL.

Each SiteSection subclass defines the name used in the left-hand menu, the prefix of all its pages, the number of pages visible in the left-hand menu when within this section, a suffix and title for each page in the section (including hidden pages), the directory that the section should be deployed under, and a resource bundle name for translations. Furthermore, the SiteSections have hooks for code that needs to be run on deployment and undeployment. If you want to add a new page to the section, you will only need to add a new line to the list of pages with a unique (within the SiteSection) suffix and a key for the page title, plus a default translation in the corresponding Translation.properties file. If you want it to appear in the left-hand menu, update the number of visible pages to n+1 and put your new pages as one of the first n+1 lines.

This is an example of what a simple SiteSection can look like. Note that only the first two pages from the list have entries in the left-hand menu. This class does no special initialisation and shutdown.

```
public HistorySiteSection() {
        super("sitesection;history", "Harveststatus", 2,
              new String[][]{
                      {"alljobs", "pagetitle;all.jobs"},
                      {"perdomain", "pagetitle;all.jobs.per.domain"},
                      {"perhd", "pagetitle;all.jobs.per.harvestdefinition"},
                      {"perharvestrun", "pagetitle;all.jobs.per.harvestrun"},
                      {"jobdetails", "pagetitle;details.for.job"}
              }, "History",
                 dk.netarkivet.harvester.Constants.TRANSLATIONS_BUNDLE);
    }

    public void intitialize() {}
    public void close() {}
```

### Processing updates

Some JSP sites cause updates when posted with specific parameters. Such parameters should always be specified in the beginning of the JSP file. All updates of underlying file systems, databases etc should happen before **generat eHeader()** is called, so processing errors can be properly redirected. The preferred way to process updates is to create a method processRequest() in a class corresponding to the web page, but under the **webinterface** package of the corresponding module. This method should take the **pageContext** and **I18N** parameters from the JSP page, together they contain all the information needed from there.

In case of processing errors, the processing method should call **HTMLUtils.forwardToErrorPage()** and then throw a **ForwardedToErrorPage** exception. The JSP code should always enclose the **processRequest()** call in a try-catch block and return immediately if **ForwardedToErrorPage** is thrown, like in the following code-fragment:

```
try {
        HTMLUtils.forwardOnEmptyParameter(pageContext,
           Constants.DOMAIN_PARAM);
        DomainConfigurationDefinition.processRequest(
           pageContext, I18N);
    } catch (ForwardedToErrorPage e) {
        return;
    }
```

This mechanism should be used for "expected" errors, mainly illegal parameters. Errors of the "this can never happen" nature should just cause normal exceptions. Like in other code, the **processRequest()** method should check its parameters, but it should also check the parameters posted in the request to check that they conform to the requirements. Some methods for that purpose can be found in **HTMLUtils**.

### I18n

We use the Apache I18n taglib for most internationalization on the web pages. This means that instead of writing different versions of a web page for different languages, we replace all natural language parts of the page with special formatting instructions. These are then used to look up translations to the language in effect in translation resource bundles.

Normal strings can be handled with the <fmt:message/> tag. If variable parameters are introduced, such as object names or domain names, they can be passed as parameters using <fmt:message key="translation.key"><fmt:param value="<%myVal>"/></fmt:message>. Note that while the message retrieved for the key gets any HTML-specific characters escaped, the values do not and should be manually escaped. It is possible if necessary to pass HTML as parameters.

Dates should in general be entered using <fmt:formatDate type="both">, though a few places use a more explicit handling of formats. This lets the date be expressed in the native language's favorite style.

Integers and longs are handled in Java properties files with {0, number, integer} or {0, number, long} as described in MessageFormat. In JSP files we convert integer/long to strings with method HTMLUtils.localiseLong(long, PageContext) or method HTMLUtil.localiseLong(long, Locale).

Note the boilerplate code at the start of every page that defines output encoding, taglib usage, translation bundle, and a general-purpose I18N object. It is important that the translation bundles from the **Constants** class for the module you're in is used, or incomprehensible errors will occur.

```
pageEncoding="UTF-8"
%><%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"
%><fmt:setLocale value="<%=HTMLUtils.getLocale(request)%>" scope="page"
/><fmt:setBundle scope="page"
basename="<%=dk.netarkivet.archive.Constants.TRANSLATIONS_BUNDLE%>"/><%
    private static final I18n I18N
            = new I18n(dk.netarkivet.archive.Constants.TRANSLATIONS_BUNDLE);
%><%
```


# Pluggable parts

**Contents**



Some points in NetarchiveSuite can be swapped out for other implementations, in a way similar to what Heritrix uses.

### RemoteFile

The RemoteFile interface defines how large chunks of data are transferred between machines in a NetarchiveSuite installation. This is necessary because JMS has a relatively low limit on the size of messages, well below the several hundred megabytes to over a gigabyte that is easily stored in an ARC or WARC file.

The RemoteFile interface is defined by the RemoteFile interface.

### JMSConnection

The JMSConnection provides access to a specific JMS connection. The default NetarchiveSuite distribution contains only one implementation, namely JMSConnectionSunMQ which uses Sun's OpenMQ. We recommend using this implementation, as other implementations have previously been found to violate some assumptions that NetarchiveSuite depends on.

The JMSConnection interface is defined by the abstract class JMSConnection.

Implementations of this interface needs to implement the four abstract methods in this interface: getConnectionFactory(),  getDestination(String destinationName), onException(JMSException e), and getQueueSession().

### ArcRepositoryClient

The ArcRepositoryClient handles access to the Archive module, both upload and low-level access.

The ArcRepositoryClient interface is defined by the interface ArcRepositoryClient

### IndexClient

The IndexClient provides the Lucene indices that are used for deduplication and for viewerproxy access. It makes use of the ArcRepositoryClient to fetch data from the archive and implements several layers of caching of these data and of Lucene-indices created from the data. It is advisable to perform regular clean-up of the cache directories.

The IndexClient interface is defined by the Java interface JobIndexCache

### Archive Admin DBSpecifics

Defines functionality specific to the type of database for the Archive Admin database, see javadoc for details.

### Harvester DBSpecifics

Defines functionality specific to the type of database used for the Harvester module, see javadoc for details.

### Notifications

The Notifications interface lets you choose how you want important error notifications to be handled in your system. Two implementations exist, one to send emails, and one to print the messages to `System.err`. Adding more specialised plugins should be easy.

The Notifications interface is defined by the abstract class Notifications.

### HeritrixController

The HeritrixController interface defines our interface for initialize a running Heritrix instance and communicate with this instance. We have two implementations that starts heritrix as its own process and then communicates with it using JMX (JMXHeritrixController, BnfHeritrixController), and a deprecated implementation with heritrix embedded inside NetarchiveSuite (DirectHeritrixController), which controls Heritrix using a CrawlController instance.

The HeritrixController interface is defined by the Java interface HeritrixController.

### ActiveBitPreservation

The ActiveBitpreservaton interface defines our interface for initializing bitpreservation actions from our GUI. We have a filebased (now deprecated) and a database based implementation. Both these implementations communicate with the archive through the ArcRepository interface.

The ActiveBitPreservation interface is defined by the Java interface ActiveBitPreservation

## XML handling by Deploy

The deploy program needs to create a settings file in the `xml` fileformat for each application.
The creation of these files involves manipulating `xml` datastructures, creating new `xml` instances based on previous ones and saving the result to a file.

The `XMLTree` class (under `dk.netarkivet.common.utils`) did not support this functionality, and it would

require a major structural change in the class to meet our demands.

A new class (`dk.netarkivet.deploy.XmlStructure`) was therefore implemented to meet our requirements for handling the `xml` datastructure.
This class uses the `com.dom4j` structure for handling the `xml` datastructure.

The `dk.netarkivet.deploy.XmlStructure` class has the ability to inherit and overwrite, which is used in the deploy configuration structure.
The abilty to inherit is implemented as creating a new instance identical to a current instance, thus inheritance can only occur during creation of a new instance of this class.

The overwrite function merges the current !XmlStructure with a new tree.
This means that the leafs which are present in both new tree and the current one, the value in the current leaf will be overwritten by the leaf on the new tree.
Branches which only exists in the new tree will be appended to the current tree.
The branches in both trees are then recursively overwritten

# Archive Design

**Contents**



The Archive Design Description contains the description of overview of how the archive works, describes Indexing and caching and describes CrawlLogIndexCache.

## Archive Overview

The NetarchiveSuite archive component allows files to be stored in a replicated and distributed environment.

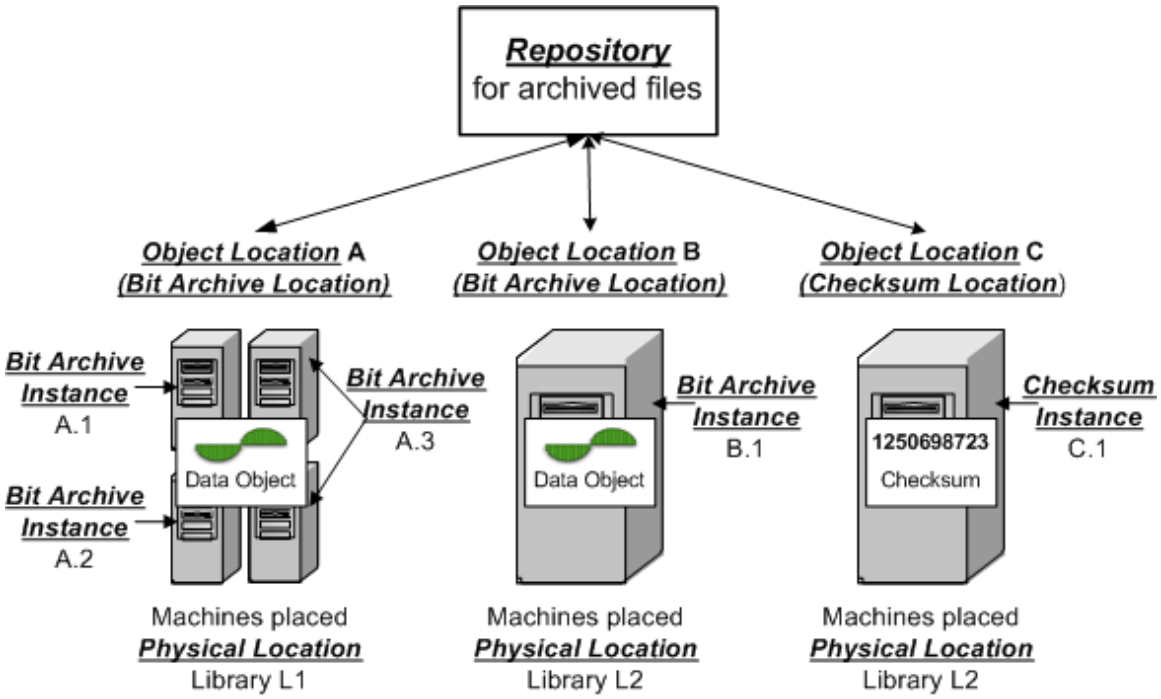
Basically, the storage nodes are replicated over a number of **bitarchive replicas**. During the storing process each copy is verified to have the same checksum, before a store is accepted. Besides that, functionality for automatically checking that each bitarchive replica holds the same files with the same checksum, ensures that ever loosing a bit is highly improbable.

For each **bitarchive replica**, the files may be stored in a distributed manner on several **bitarchives instances**. The philosophy is that you can use off-the-shelf hardware with ordinary hard disks for one bitarchive replica. This allows

you to use cheaper hardware, and get more CPU-power per byte. This may be important if you regularly want to perform large tasks on the given bits, like indexing or content analysis. Beware however, that the power usage may be higher in such a setup, and that the maintenance needs may be higher.

**Architecture**

The archive architecture is shown in the following figure:




where

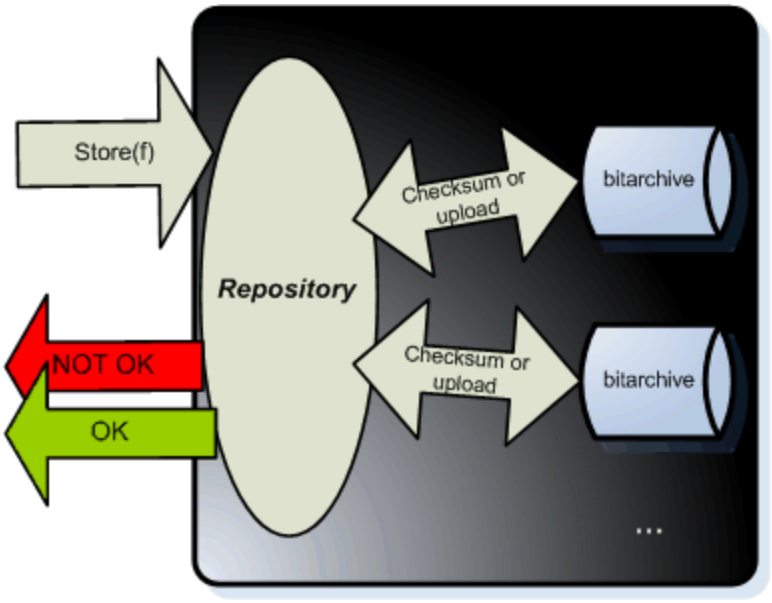
- Repository is handled by the ArcRepository application
- Replica X for bitarchives is handled by the BitarchiveMonitor application
- Bitarchive Instance X.N is handled by the Bitarchive application
  So there is

- An ArcRepository: **Exactly One**
- BitarchiveMonitors: **One per bitarchive replica**
- Bitarchives: **As many as you like per bitarchive replica**
  The components communicate using JMS, and a file transfer method (currently FTP, HTTP and HTTPS methods are implemented).

The public interface is through the ArcRepository. This interface allows you to send JMS messages to the ArcRepository and get responses, using the JMSArcRepositoryClient. It has methods to store a file, get a file, get one record from an ARC file, and run batch jobs on the archive. Batch jobs allow you to submit a job to be run on all files, or selected individual files, on a given location, and return the results. Additionally, there are some methods for recovering from an error scenario which we will cover shortly under bit preservation.

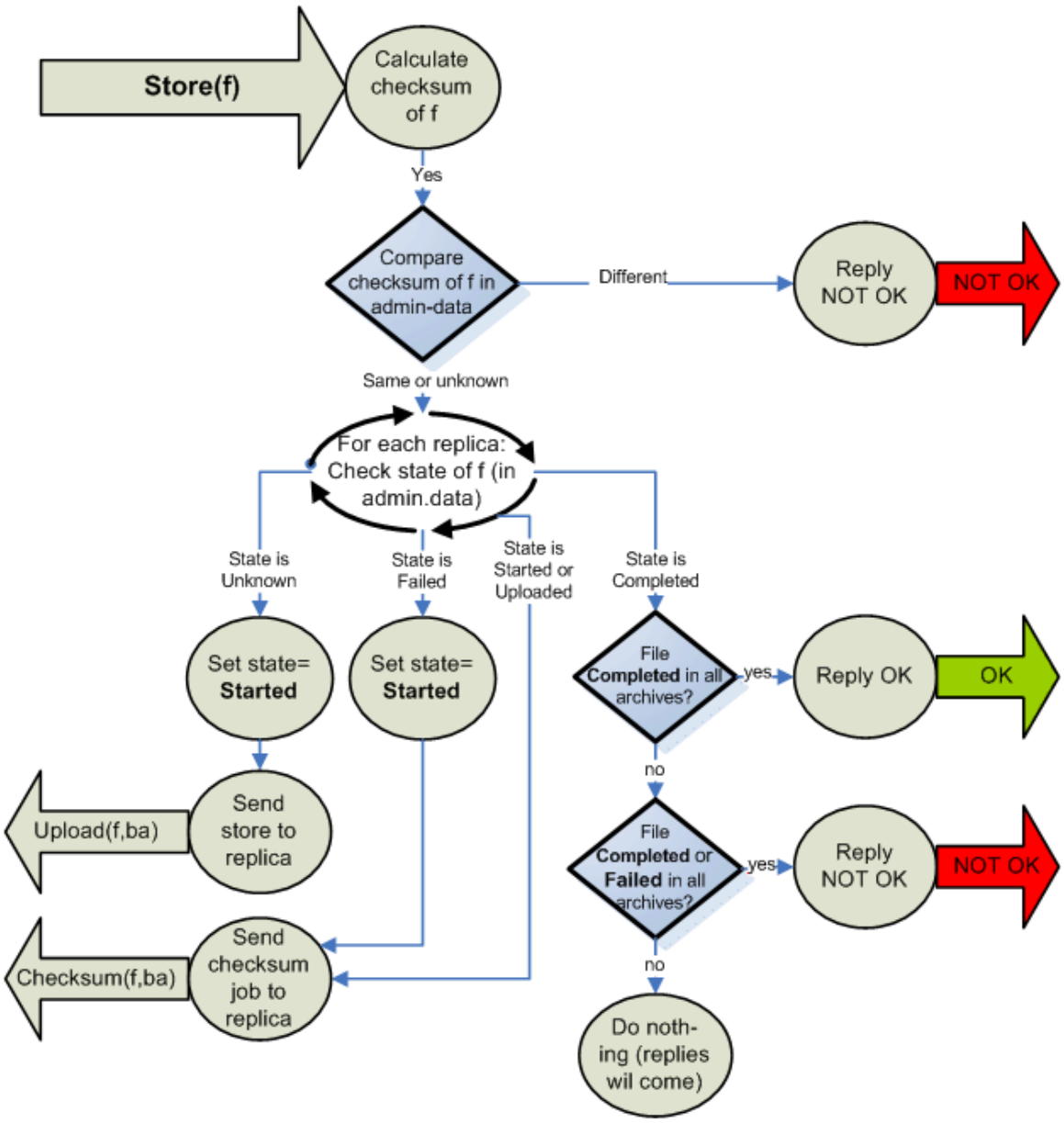
**Repository State Machine**

When files are uploaded to the repository, it uploads the file by sending upload requests and checking the uploads as sketched in the below figure.



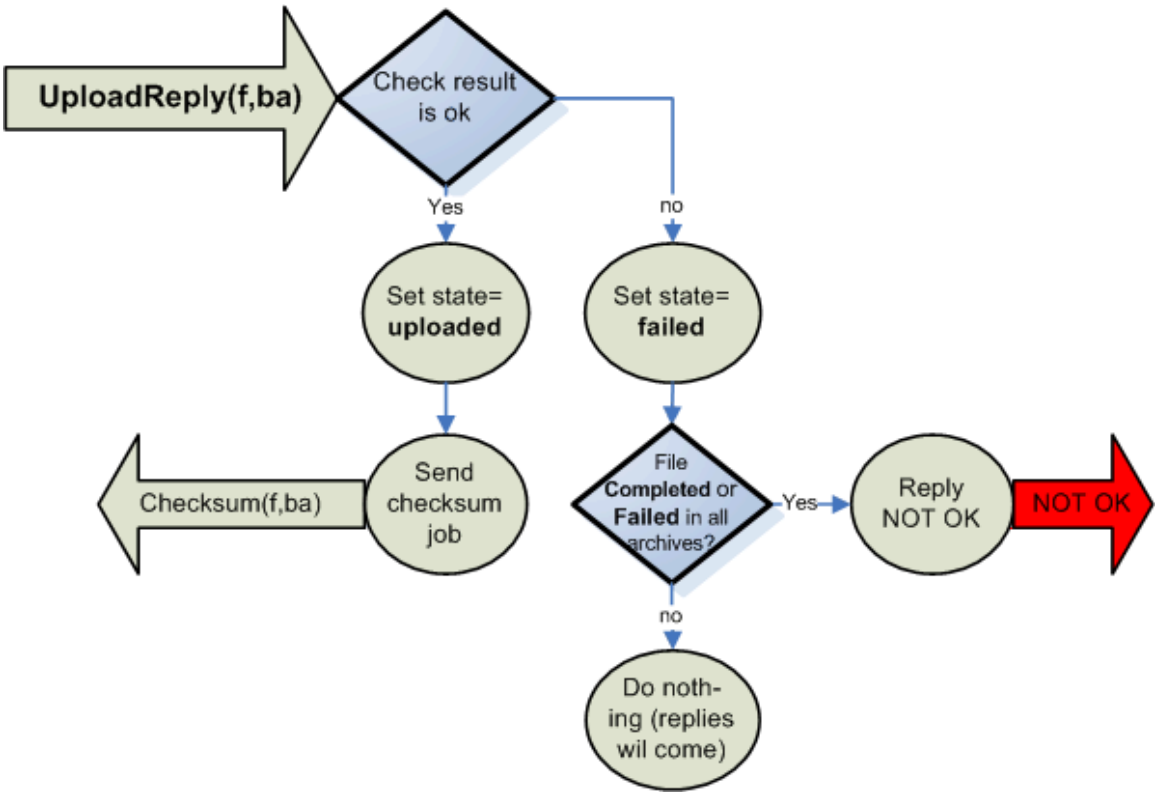

Internally this is handled following a state machine based on the messages it receives. These can either be

- a store message for a file to be stored
- a Upload Reply message from a bitarchive replica that was requested to upload a file (in storing process).
- a Checksum Reply message from a bitarchive replica that was requested find checksum as part of checking a file status (in storing process).
  The state diagram for each message is given in the below figures:

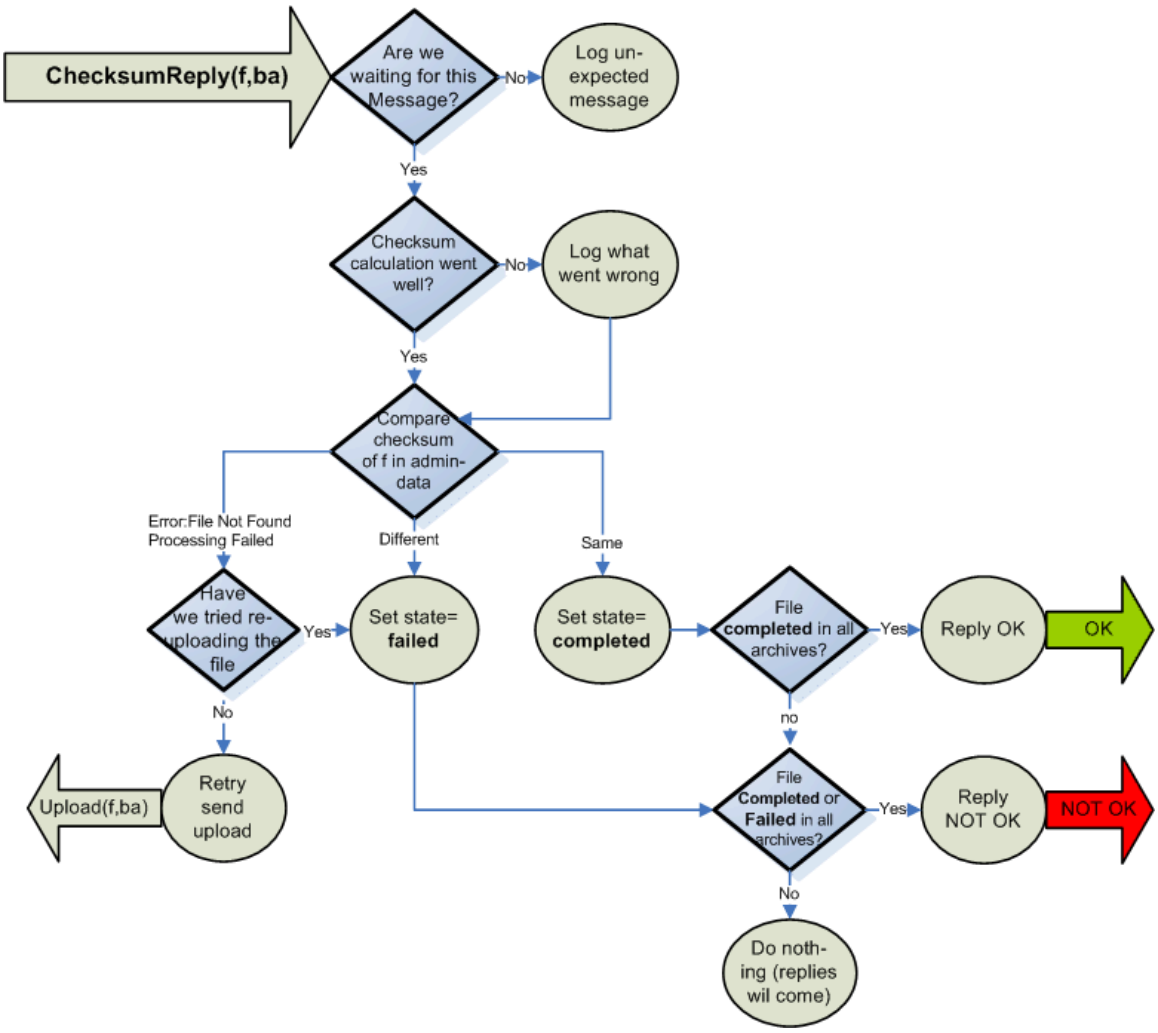



**Store message**

**Upload Reply message**



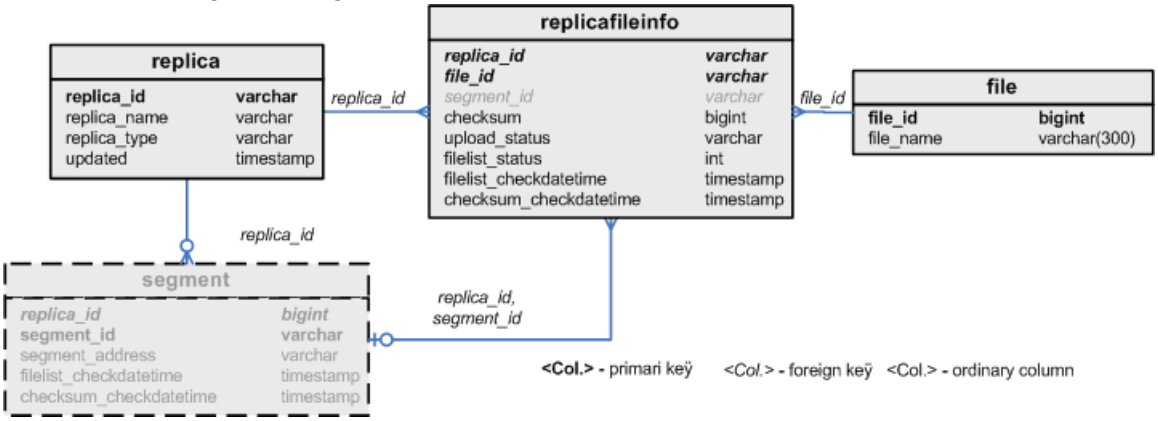

**Checksum Reply message**




***Admin for the Repository***

The ArcRepository keep a record of the upload status of all the files for all the replicas. This information was until release 3.10 stored in the admin.data file. This solution is now deprecated. Now the information about the files in the replicas is now by default stored  in a database. This database can also be used for the bitpreservation.

It has the following table diagram:

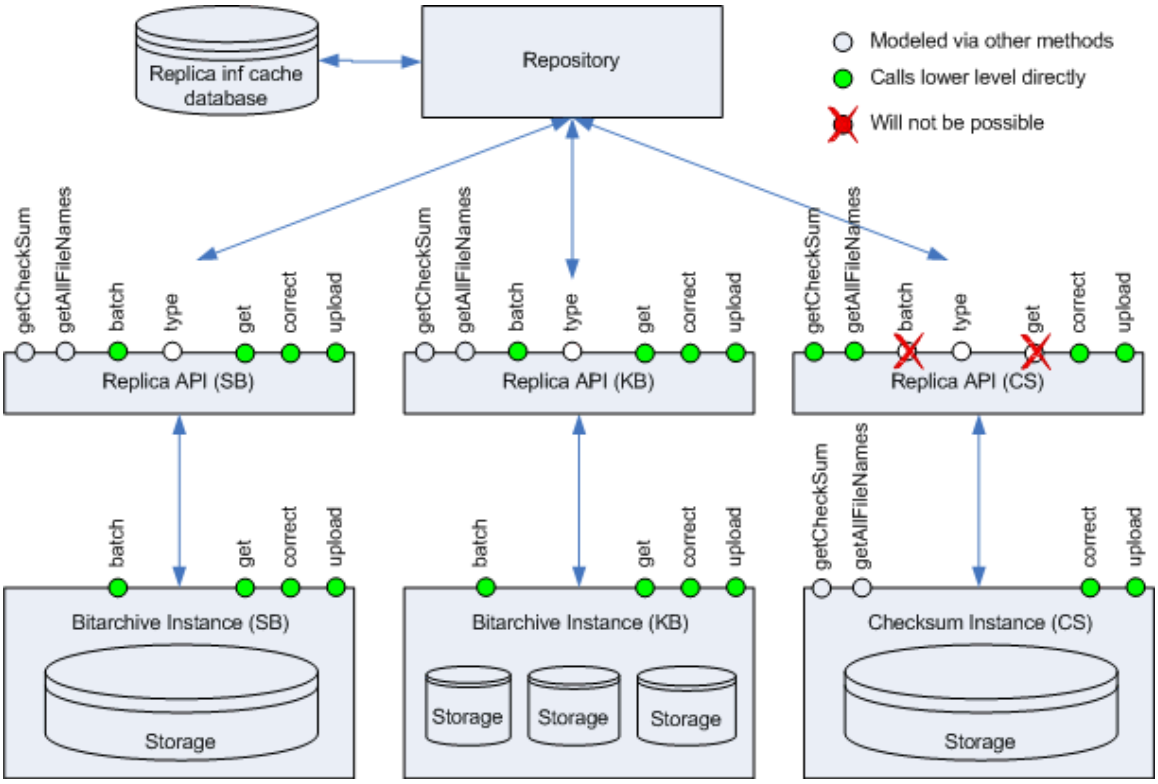



A few databases indices are also needed:

```
create index fileandreplica on replicafileinfo (file_id, replica_id);
create index replicaandfileliststatus on replicafileinfo (replica_id,
filelist_status);
create index replicaandchecksumstatus on replicafileinfo (replica_id,
checksum_status);
create index fileindex on file (filename);
```

***Communication between ArcRepository and the replicas***

The following drawing shows the message interaction between the ArcRepository and the replicas:




The 'replica inf cache database' is the database described above.

**Extra functionality**

Besides the basic architecture, the archive component contains the following extras:

- An index server, providing three kinds of indexing capabilities of a location.
- A bitpreservation web interface, providing an interface for monitoring bit
  . integrity and handling error scenarios(missing/corrupt files in the archive)
- Command line tools for uploading a file, getting a file, getting an arc
  . record or running a batch job.

**The index server**

The index server allows you to build an index over your data.

It does this by using the batch method defined in the arc repository.

It assumes you have an archive containing only ARC files, and that arcfiles are named

```
<<job-number>>*-.arc(.gz)
```

and that for each job number, there are arc files with the names

```
<<job-number>>-metadata-*.arc
```

containing the crawl.log for the harvest that generated the arc files, and a cdx file for all the arc files. These files will be generated by the NetarchiveSuite harvester component.

Using the IndexRequestClient, you may request indexes over a number of jobs, either of type CDX, or as a lucene index. The lucene index comes in two different flavours,

- One is used for the deduplication feature of the harvesting component, which only contains the objects that are not of mime-type text/*
- The other is used by the access component, and contains all objects.

**The bit preservation interface**
For monitoring the bit integrity of your archive, and for performing actions in case of bit errors, a user interface is available. This is installed as a site section of the NetarchiveSuite webserver.

This will basically give you a user interface with the following features:

- Perform a check that all files are present on a given bitarchive replica (can take hours to complete this check)
- Perform a check that all files have correct checksum on a given bitarchive replica (can take hours to complete this check)
- Reestablish files that are missing on one bitarchive replica, but available in another replica
- Replace a file with bad checksum (ie. the file is corrupt) on one bitarchive replica, where a healthy copy is available in another replica

## Indexes and caching

The deduplication code and the viewer proxy both make use of an index generating system to generate Lucene indexes from the data in the archive. This system makes extensive use of caching to improve index generation performance. This section describes the default index generating system implemented as the IndexRequestClient plugin.

There are four parts involved in getting an index, each of them having their own cache. The first part resides on the client side, in the IndexRequestClient class, which caches unzipped Lucene indexes and makes them available for use. The IndexRequestClient receives its data from the CrawlLogIndexCache in the form of gzipped Lucene indexes. The CrawlLogIndexCache generates the Lucene indexes based on Heritrix crawl.log files and CDX files extracted from the ARC files, and caches the generated indexes in gzipped form. The crawl.log files and CDX files are in turn received through two more caches, both of which extract their data directly from the archive using batch jobs and store them in sorted form in their own caches.

All four caches are based on the generic FileIndexCache class, which handles the necessary synchronization to ensure that not only separate threads but also separate processes can access the cache simultaneously without corrupting it. When a specific cache item is requested, the cache is first checked to see if it already exists. If it doesn't, a file indicating that work is being done is locked by the process. If this lock is acquired, the actual

cache-filling operation can take place, otherwise another thread or process must be working on it already, and we can wait until it finishes and take its data.

The FileIndexCache class is generic on the type of the identifier that indicates which item to get. The higher-level caches (IndexRequestClient and CrawlLogIndexCache) use a Set<Long> type to allow indexes of multiple jobs based on their IDs. The two low-level caches just use a Long type, so they operate on just one job at a time.

The two caches that handle multiple job IDs as their cache item ID must handle a couple of special scenarios: Their cache item ID may consist of hundreds or thousands of job IDs, and part of the job data may be unavailable. To deal with the first problem, any cache item with more than four job IDs in the ID set is stored in a file whose name contains the four lowest-numbered IDs followed by an MD5 checksum of a concatenation of all the IDs in sorted order. This ensures uniqueness of the cache file without overflowing operating system limits.

Every subclass to FileBasedCache uses its own directory, where the cache files are placed. The name of the final cache file is uniquely created from the id-set, which should be made into an index. Since the synchronization is done on the complete path to the cache file, then it must require two instances of the same class (e.g. DedupCrawlLogIndexCache), which is attempting to make cache on the same id-set at the same time, for a synchronisation block to occur. In this case the cache file would anyway only be made once, since the waiting instance will use the same cache file created by the first instance.

A subclass of CombiningMultiFileBasedCache uses a corresponding subclass of RawMetadataCache to make sure that an cache file for every id exists (an id-cache file). If this file does not exist, then it will be created. Afterwards all the id-cache files will be combined to a complete file of the wanted id-set.

The id-cache files are blocked for other processes during their creation, but they are only created once since they can be used directly to create the Lucene cache for other id-sets, which contain this id.

### CrawlLogIndexCache

The CrawlLogIndexCache guarantees that an index is always returned for a given request, regardless of whether part of the necessary data was available. This is done by performing a preparatory step where the data required to create the index is retrieved. If any of the data chunks are missing, a recursive attempt at generating an index for a reduced set is performed. Since the underlying data is always fetched from a cache, it is very likely that all the data for the reduced set is already available, so no further recursion is typically needed. The set of job IDs that was actually found is returned from the request to cache data, while the actual data is stored in a file whose name can be requested afterwards. Note that future requests for the full set of job IDs will cause a renewed attempt at downloading the underlying data, which may take a while, especially if the lack of data is caused by a time-out.

The CrawlLogIndexCache is the most complex of the caches, but its various responsibilities are spread out over several superclasses.

- The top class is the generic FileBasedCache handles the locking necessary to have only one thread in one process at a time create the cached data. It also provides two helper methods: getIndex() is a forgiving cache lookup for complex cache items that handles the partial results described before, and the get(Set<I>) method allows for optimized caching of multiple simple cache requests.
- The MultiFileBasedCache handles the naming of files for caches that use sets as cache item identifiers.
- The CombiningMultiFileBasedCache extends the MultiFileBasedCache to have another, simpler cache as a data source, and providing an abstract method for combining the data from the underlying cache. It adds a step to the caching process of getting the underlying data, and only performs the combine action if all required data was found.
- The CrawlLogIndexCache is a CombiningMultiFileBasedCache whose underlying data is crawl.log files, but adds a simple CDX cache to provide data not found in the crawl.log. It also implements the combine method

by creating a Lucene index from the crawl.log and CDX files, using code from Kristinn Sigurðsson. The other subclass of CombiningMultiFileBasedCache, which provides combined CDX indexes, is not currently used in the system, but is available at the IndexRequestClient level.

- The CrawlLogIndexCache is further subclasses into two flavors, FullCrawlLogIndexCache which is used in the viewer proxy, and DedupCrawlLogIndexCache which is used by the deduplicator in the harvester. The DedupCrawlLogIndexCache restricts the index to non-text files, while the FullCrawlLogIndexCache indexes all files.
  The two caches used by CrawlLogIndexCache are CDXDataCache and CrawlLogDataCache, both of which are simply instantiations of the RawMetadataCache. They both work by extracting records from the archived metadata files based on regular expressions, using batch jobs submitted through the ArcRepositoryClient. This is not the most efficient way of getting the data, as a batch job is submitted separately for getting the files for each job, but it is simple. It could be improved by overriding the get(Set<I>) method to collect all the data in one batch job, though some care has to be taken with synchronization and avoiding refetching unnecessary data.

# Harvester design

**Contents**



## Database

Description of the individual tables and their columns can be now found in the Derby SQL create script createfullhddb.sql.

## Harvesting roundtrip

This section describes what goes on during a harvest, from the templates are uploaded and harvests created till the

ARC files are uploaded to the archive and historical information about the harvest is stored.

## Initial steps

To create a harvest in the first place, we need to have a template to base it on. Additionally, we need a schedule for a selective harvest or some domains to harvest for a snapshot harvest.

### Uploading templates

Templates (Heritrix order.xml files) are uploaded using the **Definitions-upload-harvest-template.jsp** page. Templates are internally represented with the HeritrixTemplate object, which as part of its constructor verifies that certain elements - later modified programmatically - exist in the template. The template is then stored in the **templates** table in the database.

### Creating domains

Domains can either be auto-created through selective harvest definitions or mass upload, or they can be created manually through the interface. Domains are represented with a Domain object, which in turn contains a list of SeedList objects and a list of DomainConfiguration objects. These are stored in the **domains**, **seedlists**, and **configurations** tables respectively. New domains are created with a single configuration, a single seedlist with only one seed

```
http://www.<domain>
```

and a limit on the number of bytes and/or documents to download.

### Creating schedules

NetarchiveSuite comes with four schedules, repeating respectively hourly, daily, weekly and monthly. More schedules can be created using the web interface. Schedules are represented with the Schedule object and stored in the **schedules** table in the database.

## Creating a selective harvest

Harvests are created using the web-based user interface, as described in the User Manual.

## Job creation and and dispatching

This functionality is handled by the classes in the harvest scheduler package.

The HarvestJobGenerator selects all harvests that are ready to harvest, i.e. harvests that are activated and where the next harvest date is in the past (or, for snapshot harvest, they haven't run yet). For each harvest, a new thread is started to perform the operations required to send off the jobs – this keeps snapshot harvests from blocking scheduling of selective harvests. Since the threads may run for a while, we keep a set of harvests currently undergoing scheduling and uses it to avoid that the same harvest gets scheduled several times concurrently.

### Splitting

Splitting a harvest into jobs and sending those jobs off to the harvesters happens, as mentioned, in a separate thread. The first part of splitting is to reduce the harvests into chunks that can be handled by the HarvestJobGenerator itself – since the data structures for domain configurations and their associated domains

contain a fair amount of information, we cannot keep all of them in memory at the same time. For that reason alone, we split harvests into arbitrary chunks with no more domain configurations per chunk than the **configChunkSize** set ting allows. We use an iterator to avoid keeping all the domains and their configurations in memory for this operation, which iterates over all configurations in sorted order. The configurations are sorted by order.xml template, then by maximum number of bytes to harvest, and finally by the expected number of objects harvested. This makes sure that configurations that should be in the same job are sorted next to each other. A FilterIterator weeds out configurations in a snapshot harvest whose domains either are marked as aliases or were completely harvested in the snapshot harvest that the current snapshot harvest is based on.

For each chunk, we iteratively create new jobs by taking one domain configuration at a time and checking if it can be added to the job we're building. If it cannot, we store the job and start making a new one. Note that the jobs created are not submitted to the harvesters yet; that happens asynchronously as part of the dispatching check.

The check for whether a configuration can be added to a job is the most complex part of the scheduling system. It is based on the need to partition the domains into chunks such that all domains in a job take approximately the same amount of time to harvest and doesn't exceed Heritrix memory limits. The estimation of the size of a domain is complicated by the facts that previously unharvested domains have an unknown size, and that domains can easily increase in size by several orders of magnitude by adding forums, image galleries or crawler traps. Furthermore, each Heritrix instance can only use one order.xml file.

Whether a domain configuration can be added to a job is a multi-stage check with the following stages:

1. The configuration must not already have been added to the job.
2. The job must not end up with more than configChunkSize configurations.
3. The configuration must use the same crawl template as the other configurations in the job.
4. If the byte limit for this job is determined by the harvest definition, the configuration must not have a smaller byte limit than the definition specifies. If the byte limit for the job is determined by the other configurations in the job, this configuration must have the same byte limit as the other configurations.
5. The expected number of objects harvested by all configurations in the job, based on previous harvests of the configurations, must not exceed the **maxTotalSize** setting.
6. The **relative** difference between the largest and smallest expected numbers of objects harvested by configurations in the job must be no more than the **maxRelativeSizeDifference** setting. Note that the default setting for this is 100, so expectations within a job differ by a factor 100, not just 100%. This prevents jobs from finishing many small configurations quickly and take a long time to finish a few, large configurations.

- However, if the **absolute** difference between the largest and smallest expected numbers of objects harvested by configurations in the job is less than the **minAbsoluteSizeDifference** setting, the relative difference is ignored. This allows the very smallest configurations to be lumped together in fewer jobs.

**Note: Check on overrides.**

The expected number of objects is found based on previous harvests of a given configuration and a few assumptions about the development of web sites. If a configuration hasn't been harvested before, defaults from the settings file are used. Expectations for previously harvested domains are calculated as follows:

1. The "best" previous harvest to estimate from is found by picking the most recent complete harvest using the configuration, or the harvest that harvested the most objects if the configuration never completed.
2. The expected size per object is found based on the average size in the "best" previous harvest, if that harvest got enough objects to be considered (at least 50), but at least as many as the **expectedAverageBytesPerO bject** setting.
3. A maximum number of objects is found based on the current limits of the configuration and the harvest and the expected size per object. If neither configuration nor harvest imposes any limits, an artificial limit for estimation purposes is taken from the **maxDomainSize** setting.
4. A minimum number of objects is the number of objects found in the the "best" previous harvest, or is 0 if no previous harvest was found.

5. If the configuration had previously been completed, the estimated number of objects is the difference of minimum and maximum divided by the **errorFactorPrevResult** setting plus the minimum.

- Otherwise, the estimated number of objects is the difference of minimum and maximum divided by the **errorFactorBestGuess** setting plus the minimum.

1. The expected number of objects is capped by the maximum based on the limits.

The **errorFactorBestGuess** setting should generally be smaller than the the **errorFactorPrevResult** setting, since there is more uncertainty about the actual number of objects when the harvest has never been completed. These two settings are best understood as the largest possible factor of error between our estimate and reality. If we use an error-factor of 10, we accept that while configurations could end up growing by as much as the hard limits allow, we split as if they only grow by one-tenth that amount. In most cases, growth will be limited, but it is likely that if a new archive, forum or such is added to a site, the site can grow significantly between harvests. These settings determine the trade-off between the likelihood that some sites have grown a lot and the desire to keep similar-sized configurations in the same job.

Once the job does not get any more domain configurations added to it, it is added to the database with status 'New', and cannot change further except for status updates.

When all domain configurations for a harvest have been placed in jobs, the time for the next execution of the harvest is set. Note that the execution time is updated regardless of whether the jobs are actually successful, or even have been run. Additionally, the counter of number of runs is updated.

If there are any errors in the scheduling process, including the creation of jobs, the harvest is deactivated to prevent the system from being overloaded with broken scheduling attempts.

### Talking to the harvesters

New Jobs are sent to any available harvesters as a DoOneCrawlMessage . This message contains not only the Job object, but also some metadata entries that are associated with the job. Currently, the metadata consists of a listing of the aliases that were used in the job creation and of a listing of the job IDs that should be used to get the deduplication index. The harvests report back via JMS when they are ready to process harvest jobs (see JobDispatcher).

The DoOneCrawlMessages are placed on a JMS queue, either `ANY_HIGHPRIORITY_HACO` for selective/event harvests or `ANY_LOWPRIORITY_HACO` for snapshot harvests. At the same time, the job is set to status 'Submitted', indicating that it's in queue for being picked up by a harvester. The names of these queues is a historical artifact and does not indicate that "high priority" jobs can "get ahead" of "low priority" jobs, and there could potentially be just one or more than two queues. Notice that since the JMS messages are expected to be cleaned from the queues at system restart, we assume that any messages about jobs in state "Submitted" are lost after a restart, and they are therefore automatically resubmitted at system startup.

Each HarvestControllerServer application listens to just one of the two queues. When it receives a message (remember that JMS guarantees exactly-once delivery for queues), it immediately sends a message back that tells the HarvestJobManager that the job has been picked up and can be put in state 'Started'.

At this point, the HarvestControllerServer has accepted that it will attempt to run the job and can start to set up the files necessary for running Heritrix.

### Harvest setup

The directory used in a crawl is created by the HarvestControllerServer, using the job id and timestamp in the directory name. Details on what Heritrix reads and writes can be found in the Heritrix "outside the GUI" page.

### Running Heritrix

The HeritrixLauncher class sets the correct file paths in the Heritrix order.xml file and keeps an eye on the progress of the harvest. If Heritrix does not download any data for a period of time defined by the **noResponseTimeout** settin g, HeritrixLauncher will stop the crawl. This is to avoid a single very slow web server from extending the crawl for very little gain. Also, if no crawler threads are active in Heritrix for a period of time defined by the **inactivityTimeout** setting, HeritrixLauncher will stop the crawl. This is a workaround for a subtle bug in Heritrix.

Heritrix is run by the harvester system as a standalone process. This allows access to Heritrix' web interface. The interfacing to the Heritrix process is controlled by JMXHeritrixController (or the more recent BnfHeritrixController), an implementation of the HeritrixController interface. General documentation on JMX can be found as part of the Java documentation, on the Oracle JMX Technologies pages, in the JMX Accelerated Howto, and via the JMX Wikipedia page. Heritrix' documentation of its JMX interface is partially described in the now archived JMX feedback page, but can also be investigated in more depth via the Heritrix JMX command-line client, and in the source files Heritrix.java and CrawlJob.java (links for Heritrix version 1.14.4).

JMXHeritrixController (or BnfHeritrixController) starts a new process as part of its constructor, putting the jar files in **l ib/heritrix/lib** and the NetarchiveSuite jar files in the classpath. The process is started in the directory created by the HarvestControllerServer, and all files created as part of the crawl are put into that directory. Stdout and stderr from Heritrix, along with a dump of the startup environment, are put in the **heritrix.out** file. The full command line used for running Heritrix is put in the log file.

Before the process is started, a shutdown-hook is added to attempt proper cleanup in case the harvest controller is shut down prematurely. Notice that this hook is removed if the process finishes normally.

After constructing the JMXHeritrixController object, HeritrixLauncher calls the initialize() method on the JMXHeritrixController, which first checks that we're talking to the correct Heritrix instance (in case one was left over from earlier), then uses the addJob JMX command to create a job for the crawl. Before returning from initialize, we call getJobName() to extract from the job a unique name we can use to locate it by later. getJobName() also has the task to wait (using exponential back-off) until the job has actually been created, since the addJob command can return before the job actually exists.

After initialize() is done, the requestCallStart() method executes the JMX command requestCrawlStart to start the job, and we then enter a loop for the duration of the crawl. Inside the loop, we check for the two timeouts as well as for orderly termination of the job and log status reports every 20 seconds. These logs can be seen by the user in the System Overview web page.

Access to the Heritrix user interface can be had by connecting to the port specified by the **heritrixAdminGui** setting, using the admin name and password specified by the **heritrixAdminName** and **heritrixAdminPassword** settings, respectively.

The cleanup of the JMXHeritrixController involves issuing the shutdown JMX command to Heritrix, then waiting for a while (duration defined by the processTimeout setting) for Heritrix to end crawls and write its reports. If Heritrix doesn't stop within the timeout period, we forcibly kill it. After that, we collect the exit code and wait for the stdout/stderr collector processes to finish.

### Creating metadata

After the heritrix has finished with the harvesting, the harvest is documented, and the result of this documentation is stored in a separate arcfile prefixed with the job id, and ending with "-metadata-1.arc". This metadata file contains all heritrix logs and reports associated with this harvest(crawl.log, local-errors.log, progress-statistics.log, runtime-errors.log, uri-errors.log, heritrix.out, crawl-report.txt, frontier-report.txt, hosts-report.txt, mimetype-report.txt,

processors-report.txt, responsecode-report.txt, seeds-report.txt), some metadata about the job itself, and CDX'es of the contents of the arcfiles created by Heritrix. A CDX line points to where an object is located in an arcfile, its length and mimetype. This metadata arcfile is uploaded along with the rest of the arcfiles.

### Uploading

When Heritrix is finished, and the metadata arcfile created, all arcfiles are uploaded to the archive using a ArcrepositoryClient.

### Storing harvest statistics

When uploading is done, a status message is sent back to the scheduler, containing error reports and harvest statistics. Errors are split into harvest errors and upload errors, since upload is attempted even if the harvest fails. For each, a short error description and a longer, detailed description are sent. The statistics sent are the following for each domain harvested:

- Number of objects harvested
- Number of bytes harvested
- Reason the harvest stopped, one of completed (no more objects found), object-limit (hit maximum allowed number of objects), size-limit (hit maximum allowed number of bytes, as specified by the harvest), config-size-limit (hi maximum allowed number of bytes, as specified by the configuration), and unfinished (the harvest was interrupted before any of the other stop reasons applied).

...need to specify what gets counted within a domain...

...need to clarify the states of a harvest...

When the status message is received, the statistics from it is stored per domain in the database, along with the job number, the harvest number, the domain name, the configuration name, and a timestamp for receipt of the information.

After the harvest statistics have been sent to the database, the HarvestController application checks, if there is free space on the machine for a new harvestjob. If this is the case, it starts to listen on the job queue. If not, it goes into a dormant mode.

### Old jobs

When a harvester application starts up, it checks whether any jobs are left from previous runs, in case the harvest or the upload was aborted. If there is, the last three steps described above are taken for the old jobs before the harvest application starts listening for new jobs.

### Deduplication in NetarchiveSuite

deduplication is performed by using the DeDuplicator module developed by Kristinn Sigurdsson as the first of Heritrix write-processors:

```
<newObject name="DeDuplicator" class="is.hi.bok.deduplicator.DeDuplicator">
        <boolean name="enabled">true</boolean>
        <map name="filters"/>
        <string
name="index-location">/home/test/JOLF/cache/DEDUP_CRAWL_LOG/empty-cache</string>
        <string name="matching-method">By URL</string>
        <boolean name="try-equivalent">true</boolean>
        <boolean name="change-content-size">false</boolean>
        <string name="mime-filter">^text/.*</string>
        <string name="filter-mode">Blacklist</string>
        <string name="analysis-mode">Timestamp</string>
        <string name="log-level">SEVERE</string>
        <string name="origin"/>
        <string name="origin-handling">Use index information</string>
        <boolean name="stats-per-host">true</boolean>
      </newObject>
```

This uses a Lucene (v. 2.9.4) index with information about previously harvested objects. This index may be empty.

In NetarchiveSuite, the index contains entries for objects fetched in an earlier harvest job, which this harvestjob is likely to revisit, i.e. this new harvestjob revisits some of same domains that the previously harvest jobs did.

The following arc record part of the metadata for every harvestjob mentions the NetarchiveSuite harvest jobs, that this job can be seen to continue. In the following record, only job with ID 2 is mentioned:

```
metadata://netarkivet.dk/crawl/setup/duplicatereductionjobs?majorversion=1&minorvers
ion=0&harvestid=3&harvestnum=0&jobid=4 130.226.228.7 20081215100759 text/plain 1
2
```

Each entry in the index contains the URL (unNormalized), content-digest, mimetype , Etag (if available), and the origin of the entry <arcfile>,<offset>.

Only non-text fetched URLs are indexed, as only URIs with non-text mimetypes are looked up by the DeDuplicator.

During the actual crawl-time, all non-text URIs are looked up in the index, and if a match (A URI is matched, if the URI is found, and it has the same digest as the current URI) is found in the index, the URI is ignored by the ARCWriter, but an entry is written to the crawl-log that contains the reference to the original stored URI. This reference is written to the Annotations part of the crawl-line (12th part of the crawl-line): deduplicate:arcfile,offset

Generation of deduplication indices is made by merging information in the crawl-log with information in the CDX-files generated for each job at the conclusion of the harvest.

The CDX'es contain information about the contents of the arc-files generated by Heritrix, but they lack information of the deduplicated objects. CDX-files containing references also to deduplicated objects can be generated from the crawl-logs by the tools provided in the wayback module. These tools are described in the Additional Tools manual.

The merging of this information in NetarchiveSuite was necessitated by the way we do Quality Assurance of the harvestJobs, which is done on a job by job basis, so we needed a way to refer to the deduplicated objects.

# ViewerProxy Design

**Contents**



This section describes the viewerproxy control resolver, the special viewerproxy access via urls and the observer resolver.

The viewerproxy uses the Jetty HTTP server library to handle connections. Each incoming URL is sent through a pipeline of "resolvers", each of which can either process the URL or pass it on to the next resolver. Each resolver must extend the abstract class **CommandResolver**. The **executeCommand** method must be overridden to handle requests, and should return true if the requests was handled by this resolver. The resolver is responsible for calling **response.setStatus** to set the appropriate HTTP result code.

Incoming URLs are handled by the resolvers in the following order: Viewerproxy control resolver, GetDataResolver, NotifyingURIResolver.

## Viewerproxy control resolver

The HTTPControllerServer class manages index setup and missing URL collection for the viewerproxy. It is mainly used through the QA web interface. It has the following commands:

- start recording URIS (**/startRecordingURIs**)
- stop recording URIS (**/stopRecordingURIs**)
- clear collected URIS (**/clearRecordedURIs**)
- get collected URIS (**/getRecordedURIs**)
- change index (**/changeIndex**)
- get status (**/getStatus**)

## Special Access

### Via URLs

The GetDataResolver class provides some special URLs in the viewerproxy that can be used for more direct access to the stored data. To use them, your browser must be set up to access the viewerproxy in the same way as when browsing harvested data. The general format of the commands are `http://viewerproxy.invalid/`<command>`?arg1=value1&arg2=value2`…

The commands are:

- getFile - gets a whole file from the archive
- arcFile=<filename> - name of the file (without pathnames)
- getRecord - gets a single ARC record from the archive
- arcFile=<filename> - name of the file to look up a record in (without pathnames)
- arcOffset=<offset> - offset into the file the record starts at
- getMetadata - gets all metadata for a job from the archive

- jobID=<id> - ID (numeric) of the job for which to fetch metadata

#### Via Command Line

Futhermore it is possible to make getFile and getRecord commands from the command line the following way:

```
usage: java dk.netarkivet.archive.tools.GetFile <filename> [destination-file]
```

This tool retrieves a file from the archive. If the **destination-file** is omitted, the file is stored with the same name. The bitarchive replica the file is retrieved from is chosen based on the setting **settings.archive.common.useReplicaId**.

```
usage: java dk.netarkivet.archive.tools.GetRecord <indexdir> [uri]
```

This tool depends on the existence of a luceneindex, as generated by the index server. It will use this index to lookup the arcfile and offset to get a particular record from. It will then retrieve that record from the archive.

The bitarchive replica the file is retrieved from, is chosen based on the setting *settings.archive.common.useReplicaId. The result is printed to stdout.

### Observer resolver

The NotifyingURIResolver class provides means for logging what users access through the viewerproxy. It never processes any URLs itself, merely allows a URIObserver to monitor the URLs. It is currently used to record URLs that are not handled by other resolvers.